Optimizing Memory Translation Emulation in Full System Emulators

XIN TONG, University of Toronto

TOSHIHIKO KOJU and MOTOHIRO KAWAHITO, IBM Research, Tokyo

ANDREAS MOSHOVOS, University of Toronto

The emulation speed of a full system emulator (FSE) determines its usefulness. This work quantitatively measures where time is spent in QEMU [Bellard 2005], an industrial-strength FSE. The analysis finds that memory emulation is one of the most heavily exercised emulator components. For workloads studied, 38.1% of the emulation time is spent in memory emulation on average, even though QEMU implements a software translation lookaside buffer (STLB) to accelerate dynamic address translation. Despite the amount of time spent in memory emulation, there has been no study on how to further improve its speed. This work analyzes where time is spent in memory emulation and studies the performance impact of a number of STLB optimizations. Although there are several performance optimization techniques for hardware TLBs, this work finds that the trade-offs with an STLB are quite different compared to those with hardware TLBs. As a result, not all hardware TLB performance optimization techniques are applicable to STLBs and vice versa. The evaluated STLB optimizations target STLB lookups, as well as refills, and result in an average emulator performance improvement of 24.4% over the baseline.

Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics—*Performance measures*; D.3.4 [Programming Languages]: Processors—*Runtime environments*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Full system emulator, dynamic address translation, TLB

ACM Reference Format:

Xin Tong, Toshihiko Koju, Motohiro Kawahito, and Andreas Moshovos. 2014. Optimizing memory translation emulation in full system emulators. *ACM Trans. Architect. Code Optim.* 11, 4, Article 60 (December 2014), 24 pages.

DOI: <http://dx.doi.org/10.1145/2686034>

1. INTRODUCTION

A full system emulator (FSE) is a piece of software that emulates an entire machine including the processor, memory, and devices. An FSE emulates a *guest* machine over a potentially different *host* machine. Some well-known FSEs include QEMU [Bellard 2005], Bochs [Lawton 1996], IBM system Z PDT [Ogden 2013], Android Emulator [Android Developers 2011], and Windriver Simics [Magnusson et al. 2002]. FSEs are valuable in many contexts. FSEs can serve as application development platforms where hardware is not available. They can accelerate system development by making it easier to detect, recreate, and repair flaws, especially for kernel-level software components such as kernel plug-ins or drivers [Chen and Chen 2013]. In addition, they can be used to closely monitor executions of applications to identify potential security breaches, which are sometimes difficult to track in real hardware [Portokalidis et al. 2006]. Finally,

Authors' address: 35 St. George Street, Toronto, ON, Canada.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/12-ART60 \$15.00

DOI: <http://dx.doi.org/10.1145/2686034>

Fig. 1. Memory emulation in an FSE.

they can be used to study application behavior or as building blocks of full-timing simulators in computer architecture research and development (e.g., Patel et al. [2011]; Ardestani and Renau [2013]). This work targets QEMU emulating x86 hardware. However, the techniques presented should be applicable to other hardware architectures, and possibly emulators.

FSEs are typically 10 to 100 times slower than real machines [Bellard 2005] because actions that would normally execute directly in the guest hardware are emulated in software by the FSE. As a result, an FSE often has to execute multiple host instructions to emulate a single guest instruction. Some of these instructions are used for dynamic address translation (DAT)—that is, to translate guest operating system virtual addresses (VAs)/physical addresses (PAs) to host VAs. As Figure 1 shows, DAT comprises two steps in QEMU. In the first step, the guest VA/PA is translated into the guest PA using the page table of the current running process. The page table is pointed to by the emulated x86 CR3 register in this case. In the second step, the guest PA is translated into the host VA by adding a constant offset, as the emulated memory is always backed by a contiguous region in the host VA space. Given that memory accesses are relatively frequent, an FSE ends up spending a considerable fraction of time just for DAT. Accordingly, DAT acceleration methods can greatly improve overall FSE performance. For this purpose, FSEs often use a software translation lookaside buffer (STLB) structure that caches translations from guest VAs/PAs to host VAs directly, collapsing the two steps required to translate a guest VA into one.

Implemented using a hashtable, the STLB is maintained by the emulator and searched using highly optimized software code. However, even with an STLB, DAT still consumes a considerable fraction of time in modern FSEs. Specifically, this work measures that QEMU, an industrial-strength FSE, spends about 38.1% of its execution time performing DAT. Accordingly, the goal of this work is to propose techniques for improving STLB performance.

Table I. STLB Optimizations

Optimization	Purpose
Adjusting STLB size	Reduce STLB misses
Bump-the-STLB allocation	Reduce STLB flush time
Dynamically resized STLB	Reduce STLB misses
Victim STLB	Reduce STLB misses
Translation coalescing	Fewer STLB lookups
Set-associative STLB (three variants)	Reduce STLB misses
SIMD set-associative STLB lookup	Faster STLB lookups
Secondary superpage STLB	Reduce superpage page table walks
Primary superpage STLB	Reduce pressure on main STLB
Profiling-based primary superpage lookup	Accelerate lookups for superpages and regular pages

There are three STLB actions that consume most of execution time: (1) STLB lookups, (2) STLB refills, and to a lesser extent (3) STLB flushes. Since every memory read or write has to look up the STLB, the STLB search code accounts for a significant portion of the overall execution time of an FSE. Moreover, every STLB miss triggers a guest page table walk for locating the appropriate translation, which is then cached in the STLB. This STLB refill process can take hundreds of host instructions, and more so if nested page walks in the context of virtualization are required [Bhargava et al. 2008]. Finally, when the STLB does not contain address space identifiers (ASIDs) [Jacob and Mudge 1998], the STLB must be flushed when switching across processes in the guest operating system.

To reduce the amount of time spent in DAT, this work analyzes the behavior of DAT in QEMU and investigates several techniques for improving the performance of STLB lookups and refills. Table I enumerates the techniques evaluated. Some of the techniques are motivated by corresponding optimizations for hardware TLBs. This includes the set-associative STLB and the victim STLB. Others, such as the dynamically resized STLB, profiling-based superpage lookup, and translation coalescing, are enabled by the flexibility provided by a TLB implemented in software.

In summary, this work makes the following contributions:

- (1) As its primary contribution, it measures, using hardware performance counters, where time goes in an industrial-strength FSE and over a wide range of benchmarks. Memory emulation is found to be the most time-consuming FSE component. Further measurements break down where time goes during memory emulation, and this analysis forms the basis for proposing performance optimizations for each component.
- (2) It examines the applicability of hardware-inspired TLB optimizations to software emulated TLB in the FSE.
- (3) It leverages the flexibility of a software TLB to propose innovative and effective memory emulation performance improvement techniques: dynamically sized STLB, profiling-based superpage lookup, and translation coalescing.
- (4) It implements the proposed optimizations and demonstrates that they improve overall performance by an average of 24.4% and as much as 48.0% for a memory-intensive workload.

The rest of this article is organized as follows. Section 2 analyzes where time goes during emulation in QEMU and proceeds to review how STLB lookups and refills are implemented. Section 3 evaluates several STLB refill overhead and lookup reduction techniques. Specifically, it evaluates performance trade-offs with larger STLBs, bump-the-STLB allocation, dynamically adjusting STLB size, using a victim STLB, using a set-associative STLB, coalescing multiple translations, and using a superpage STLB.

The bump-the-STLB allocation technique uses a pool of multiple STLBs and a separate thread to take STLB flushing off the critical path. The victim STLB uses a small, fully associative STLB to capture some of the conflict misses of the main direct-mapped STLB. The set-associative STLB changes the main STLB so that it contains multiple entries per set in an effort to reduce conflict misses. Several variants of set-associative STLBs are considered, including one that uses SIMD instruction extensions of recent Intel processors to accelerate searching. Translation coalescing services sequences of accesses in same translation block with a single STLB access. Finally, the superpage STLB caches translations for superpages that otherwise are treated as a collection of adjacent regularly sized pages in QEMU. Profiling-based superpage lookup further improves superpage handling by dynamically profiling and associating each emulated memory instruction with an STLB lookup sequence optimized for the page size it most likely uses. This avoids how often multiple probes, each with a different page size, need to take place to find the corresponding translation. Section 4 considers combinations of the previously considered techniques, identifying which performs best. Finally, Section 5 summarizes the findings of this work.

2. WHERE DOES TIME GO IN AN FSE?

To optimize DAT, it is important to understand where time is spent in FSEs as well as how much time is spent in DAT. Accordingly, this section reviews the operation of the STLB in QEMU and then proceeds to study STLB performance experimentally. The experimental analysis identifies the sources of STLB performance inefficiency, and these results motivate the performance optimizations proposed later in the article.

2.1. Baseline STLB

Every guest VA is translated to a host VA through the guest process page table and the emulated memory offset. To obviate the need to walk the guest page table on every translation, QEMU uses an STLB to speed up DAT. The STLB stores the offset of the guest VA/PA to host VA. When translating a guest VA to a host VA, QEMU searches the STLB table first. If there is a matching entry in the STLB, QEMU adds this offset to the guest VA to get the host VA directly. Otherwise, QEMU performs an STLB refill, where it walks the page table of the current guest process and installs the information necessary for the translation into the STLB table. In the current QEMU implementation, the STLB is a hashtable of 256 entries. There is a separate STLB per mode, where a mode indicates the current memory translation mode in which the processor is running and thus the way in which addresses should be translated (e.g., whether the process is running in user space or kernel space). When emulating the x86 architecture, there are three modes and hence three STLBs. The default index for the STLB uses bits [19:12] of the guest VA.

In addition to memory address translation, the STLB also accelerates the process of dispatching to I/O emulation functions for memory-mapped IO regions. An STLB translation entry identifies, using bits in the page offset, whether a page is backed by host memory or whether it is a memory mapped page. If an access is identified as a memory mapped page, QEMU dispatches this access to the registered I/O emulation functions.

In QEMU, there is no ASID field in the STLB entries. As a result, the STLB needs to be flushed on process switches. The performance overhead of flushing the STLB is proportional to its size. The smaller the STLB, the lower the overhead for flushing it. However, as this work shows, depending on the workload, the larger the STLB, the fewer the STLB refills. Accordingly, an implementation must carefully choose the STLB size to balance the costs of STLB flushes versus STLB refills. STLB flushes could be avoided by incorporating ASIDs into the STLB. However, this approach is

Table II. Workloads

Benchmark	Input	Purpose
KernelBuild	Building the 3.12.9 Linux kernel	x86_64 GCC (make -j1) kernel build; application development.
DACAPO [Blackburn et al. 2006]	DACAPO default input, 64-bit JVM and -XX:+UseLargePages (enable superpage for Java heap) to evaluate superpage STLB in Section 3.7	Application development
SPEC INT 2006 [Henning 2006]	Train and 64 bit compiled	Application development
KernelBoot	Booting 2.6.31.4 x86_64 Linux kernel	Multiprogrammed; application development
Multiprogrammed Workload Mixes	Dacapo default SPEC train	Multiprogrammed; application testing

Table III. Multiprogrammed Workload Mixes

Workload Mixes	STLB Sensitivity
401.bzip2+429.mcf	High + High
h2+pmd	Medium + Medium
403.gcc+456.hmmmer	Low + Low
403.gcc+429.mcf	Low + High
pmd+456.hmmmer	Medium + High

not free of trade-offs. Specifically, additional instructions would be needed to look up the ASIDs during STLB lookups and to install them during STLB refills, adding to the STLB overhead. Increasing instruction count for STLB lookups would slow down all emulated memory references. Moreover, QEMU is built to support many operating systems and architectures, many of which do not utilize ASIDs.

The rest of this section measures the amount of time spent in STLB operations and then reviews the implementation of these operations so that inefficiencies can be identified and rectified.

2.2. Experimental Methodology

This section presents a detailed execution time breakdown of the emulator using hardware performance counters on a wide range of benchmarks. Since FSEs are most often used to develop applications, this work focuses on workloads representative of common application development and testing scenarios. Table II lists the workloads used, which include building or booting a Linux kernel, SPEC INT 2006, and Dacapo. In addition, multiprogrammed workloads combining applications from SPEC INT 2006 and Dacapo are also used, as Table III shows. The Linux kernel used in this work does not flush the STLB unless the thread that is context-switched in runs in a different VA space than the thread that is context-switched out; therefore, multiprogrammed workloads are used. To create these multiprogrammed workloads, the applications are first categorized according to their sensitivity to STLB performance. Three categories of sensitivity are used: high, medium, and low. Then, mixes are created contain applications from these categories. Each mix is chosen to represent a point along the spectrum of possible multiprogrammed workload mixes, with the two extremes being a mix containing applications that are all highly sensitive to STLB performance and the other extreme containing applications that are little sensitive. All measurements are taken with the configurations detailed in Table IV.

In QEMU, an STLB lookup snippet is generated for every emulated load or store instruction. This is processor instruction cache unfriendly and wastes QEMU emulation code cache space, as most of the generated STLB lookup code is identical. This

Table IV. System Configuration

System	Configurations
Host machine	Intel [®] Core i7-4770 Haswell architecture [Jain and Agrawal 2013]. 16GB RAM. Ubuntu Linux 2.6.34.7. Performance taken with OProfile 0.9.6 [Levon 2004] and code cache monitored by the JVMTI extension [JVM JVMTI 2005].
Emulator	Emulator QEMU 1.7.0 (latest stable). Compiled with GCC 4.5.0. -O3, carrying autovectorization that is beneficial for STLB flushing. Turned on all available hardware prefetchers, as prefetchers are beneficial to STLB flush code.
Emulated machine	1 emulated x86_64 CPU, 1GB RAM. Ubuntu Linux 2.6.38 kernel. Baseline STLB: 256 entries, directly mapped TLB for each mode.

Fig. 2. STLB lookup inline versus outline.

work improves the baseline STLB implementation by outlining these STLB lookup snippets and generating calls to a common instruction sequence for all emulated memory instructions. As Figure 2 shows, the outlined STLB outperforms the inlined STLB on all studied workloads and by an average of 4.2% and a maximum of 6.7% in DACAPO eclipse. This is a result of better instruction cache performance as well as fewer code cache flushes. This outlined STLB is used as a baseline for all measurements. Accordingly, the benefits reported would be higher when compared with the stock QEMU.

2.3. FSE Execution Time Breakdown

Figure 3 reports a breakdown of total execution time for QEMU. Since the focus of this work is on STLB performance, execution time is broken down into four categories: (1) STLB lookups; (2) STLB refills; (3) emulation code cache; and (4) others, which includes the time taken to translate the guest code, look up the next translation block, handle interrupts, flush the STLB on context switches, and so forth. OProfile [Levon 2004] is used to measure the amount of time spent in each component of QEMU. The overall time spent in STLB refills is calculated as the sum of the time spent in all QEMU functions used to walk the page table and to refill the STLB. By default, OProfile lumps all generated emulation code in a single “unknown” area regardless of whether it is used for an STLB lookup or for instruction emulation. Modifications were made to appropriately measure the time spent in STLB lookups. Specifically, to

Fig. 3. QEMU emulation time breakdown.

report the time spent in the STLb lookup separately from the rest of the code cache, the JVMTI extension [JVM JVMTI 2005] was dynamically linked into QEMU and all of the outlined STLb lookup snippets were registered with the JVMTI extension such that they appear as a separate category different from the emulation code cache. As Figure 3 shows, 13.2% of the time is spent in STLb lookups, and 24.9% is spent in STLb refills on average. STLb operations take more of the overall time for benchmarks with relatively large memory footprints compared to the STLb size. For example, 62.7% of the time is spent in STLb refills in the 429.mcf benchmark. On average, 36.7% of the time is spent in the code cache, and 25.2% is spent in the “others.” In kernel boot, 38.3% of the time is spent in “others,” which is a result of a large amount of guest instructions with low reuse probability that need to be translated as the kernel boots.

Having established that STLb lookups and refills constitute a significant fraction of the overall execution time in an FSE, the next sections detail how these operations are implemented. This discussion forms the basis on which performance inefficiencies will be identified and exploited.

2.4. Dissecting STLb Refills

Measured by running QEMU over Intel’s PIN [Luk et al. 2005], an STLb miss including walking the four-level page table in Linux takes 457 x86 instructions on average. To understand why the refill takes hundreds of instructions, the following enumerates the steps necessary¹ to complete an STLb refill in QEMU:

- (1) Set up context for the page table walk (e.g., STLb miss address, size of access).
- (2) Exit the code cache.
- (3) Determine whether the STLb miss address is backed by memory page(s), not IO mapped pages.
- (4) Check for cross page accesses.
- (5) Decide how to walk the page table (e.g., is paging enabled? is x86 address extension enabled).
- (6) Walk the page table to get guest VA to PA translation while checking for permission violations.
- (7) Check whether the translation can be put into the STLb (e.g., memory watched for debugging purposes and some of the self-modifying code translations are not

¹This is not an exhaustive list of the steps required to walk the page table.

Fig. 4. STLb miss rate.

placed into the STLb so that they will trigger a TLb miss and thus be handled appropriately by the emulator runtime).

- (8) Refill the STLb structure.
- (9) Complete the STLb miss load or store operation and return to the next instruction in the code cache.

The STLb refill code implementation comprises several functions. At runtime, the STLb implementation exhibits a function call depth that exceeds five deep and thus requires a nontrivial amount of host register saves and restores. Compared to STLb lookups (covered in Section 2.5), an STLb refill is much more expensive. Therefore, even when the STLb miss rate is low, STLb refills end up taking much of the total emulation time. Figure 4 shows that the average STLb miss rate is 3.34%, with the highest being 8.09% in 429.mcf. Even though the STLb miss rate is low, as much as 62.7% of the total execution time is spent in STLb refills. As seen in Figure 3, over all benchmarks measured, STLb refills account for 24.9% of the total execution time. These results demonstrate that improving STLb refill performance has the potential to improve overall emulation performance considerably.

2.5. Dissecting STLb Lookups

As Figure 5 shows, the STLb lookup takes *nine* x86 instructions. Since an STLb lookup is needed for every emulated memory reference, this process poses a significant performance penalty. As Figure 3 has shown, STLb lookups account for 13.2% of the total emulation time on average. Accordingly, improving STLb lookups has the potential to improve overall emulation time considerably.

It is possible to utilize the host's hardware TLb to assist the translation process by installing what is in the STLb into the host hardware TLb when the emulator is running. An average 59.0% performance improvement on SPEC INT 2006 has been reported with such a technique [Chang et al. 2014]. However, the applicability of this technique is limited. Specifically, this technique requires the following: (1) one must be able to change the emulator process page table in the host operating system kernel, and (2) a 32-bit guest address space and a 64-bit host address space must be used to accommodate the FSE so that it will not interfere with the guest processes and operating system [Chang et al. 2014]. Furthermore, it is sometimes desirable, such as in architectural simulations [Tong et al. 2013], to be able to track every memory access and the PAs to which it translates.

Fig. 5. STLB lookup: Emulated STLB walk.

3. IMPROVING STLB REFILLS

Since STLB refills accounts for about 24.9% of the total emulation time on average, this section considers several techniques for reducing STLB refill time. Some of the techniques are motivated by commonly used hardware TLB optimizations, whereas others are specific to software TLB implementations.

3.1. Using a Larger STLB

Hardware TLB sizes have remained relatively small due to low access time constraints, hardware space and power limitations [Hennessy and Patterson 2003]. An STLB is implemented as a hash table in memory. A larger hash table would still require the same number of instructions to access. This raises the question whether increasing the STLB size is a straightforward and practical way of reducing the number of page table walks.

Unfortunately, there is a trade-off at play. QEMU's STLB needs to be flushed on every context switch due to the lack of ASIDs. The larger the STLB, the longer it takes to flush. The relative overhead of STLB flushes is further exacerbated because guest context switches are timed using the host clock. This results in much fewer guest instructions executed per context switch compared to real hardware. As Figure 6 shows, most workloads manage to execute a few millions of instructions per context switch, whereas kernel boot and the multiprogrammed mixes execute a lot less. The lowest progress per time quantum is observed for the kernel boot workload that manages to execute only 98K instructions per context switch. This is a result of the guest operating system giving equal amount of time to multiple processes running in different address spaces. At the end, there is a trade-off between STLB size and performance. The smaller the STLB, the lower the overhead of STLB flushes. However, the smaller the STLB, the higher the STLB miss rate, and hence the more frequent the STLB refills.

To find the best STLB size for the workloads studied, this work experimented with STLBs with 256, 4K, and 64K entries, referred to as STLB^x, where x is the STLB entry count. As Figure 7 shows (Section 3.3 describes the STLB^{ds} technique), there is no single size that performs the best for all workloads. This is a result of different working set sizes. Overall, increasing the STLB size to 4K entries from 256 entries improves performance by 15.9% on average, with 429.mcf benefiting the most by 46.1%. However, further increasing the STLB size to 64K entries results in an average performance

Fig. 6. QEMU emulated instructions per context switch.

Fig. 7. Performance improvement with STLBs of different size.

degradation relative to STLB^{4k} . STLB^{64k} spends considerably more time in flushing. For example, performance for kernel boot is only 0.54 times the baseline with STLB^{64k} since it context switches much more often. Similarly, all multiprogrammed benchmarks suffer as the size of the STL increases, because context switches happen much more often compared to the single-programmed workloads.

3.2. Bump-the-STLB Allocation

Whereas using a larger STL reduces refill overhead, it increases the flushing overhead on context switches. The bump-the-STLB technique aims at reducing this flushing overhead by taking the flushing process off the critical path. It does so by using a new STL on a context switch. This is similar to the bump-the-pointer allocation used in memory allocation systems. Initially, the system allocates a number of STLs (eight in the evaluated system) that are empty and places them in a pool. When a process starts, it gets an empty STL from this pool. On a context switch, another STL is taken from this pool. At the same time, a separate thread is used to flush the used STL and then return it, eventually, back to the pool. As a result, the main emulation thread does not have to wait to flush the STL, and the pool is eventually replenished. As an added benefit, this approach improves the microarchitectural behavior for the

Fig. 8. Emulation thread versus flush thread time.

emulation thread. In this work, the STLB flushing thread is implemented using a *pthread*, and several *pthread_mutex_t* and *pthread_mutex_trylock* are used to guarantee that only one thread has access to each STLB structure at a time. The flush thread goes to sleep on a condition variable, *pthread_cond_t*, when there are at least four free STLBs. It gets woken up when there are fewer than two empty STLBs remaining. As Figure 8 shows, the TLB flush thread accounts for only 1.1% of the time of the emulation thread for single-programmed workloads. However, for multiprogrammed workloads, the average relative flush time to emulation time is 24.2% with the maximum of 74.6% in kernel boot. Therefore, bump-the-STLB allocation can be useful when there is an abundance of physical processors or when context switches happen infrequently. As Figure 6 shows, context switches are infrequent for the single-programmed benchmarks. However, for multiprogrammed benchmarks, especially kernel boot, which executes many short running programs, context switches occur roughly every million or fewer instructions. As a result, the FSE can spend a nontrivial amount of time flushing the STLBs.

With bump-the-STLB allocation, the STLB structure is no longer allocated at a fixed memory address. Therefore, one additional instruction is generated for every STLB lookup as shown in Figure 9. Figure 10 reports performance with the proposed STLB allocation technique and for STLBs of various sizes. There is a small performance degradation of 1.3% with a 256-entry STLB and bump-the-STLB allocation denoted by $STLB^{256+bts}$. This is the result of increasing the STLB lookup path by one additional instruction while not getting enough benefit from offloading the less frequent, and relatively inexpensive for this size, STLB flushes. On the other hand, bump-the-STLB allocation unlocks the potential of the $STLB^{64k}$ denoted by $STLB^{64k+bts}$, which now improves performance by 20.6% over the baseline $STLB^{256}$ on average. This result is in stark contrast to the results of the previous section, where the flushing overhead overwhelmed performance with the $STLB^{64k}$. There are two reasons why performance is significantly better now: flush overhead is reduced and microarchitectural benefits are gained by flushing the STLBs on a separate thread. Although bump-the-STLB allocation greatly reduces the time required to flush the STLB, which is more important for larger STLBs, it does require one additional thread to flush the STLB. Accordingly, it increases pressure on execution resources when context switches happen often, as was seen, for example, in the kernel boot and the 403.gcc+429.mcf multiprogrammed workload.

Fig. 9. Bump-the-STLB allocation TLB walk.

Fig. 10. Bump-the-STLB allocation performance improvement.

3.3. Dynamically Sized STLB

Section 3.1 has shown that there is not a single STLB size that works best for all workloads. Moreover, although not explicitly shown, it is reasonable to expect that applications go through phases, each with different STLB demands. To better fit application demands, the STLB can be dynamically resized.

There can be numerous policies for adjusting the STLB size. This work investigates a utilization-based approach. When the STLB is getting close to full, the emulator doubles its size, and when the STLB is underutilized, the emulator halves its size. To estimate the current load of the STLB, the emulator uses a *load counter* that is checked on context switches. Whenever an STLB translation is installed into an empty STLB entry, the load counter is incremented, whereas it is cleared just after context switches. This adds minimal performance overhead to the STLB refill path. This work doubles the size of the STLB when its load, expressed as $\frac{\# \text{ used TLB entries }}{\# \text{ total TLB entries }}$, is higher than 70% and

Fig. 11. Emulated dynamically sized STLB walk.

halves the STLB when its load is lower than 40%. Changing the STLB size requires changing the hash indexing function, an action that requires rebuilding the hash table. For this reason, adjusting the STLB size is done on context switches where the STLB is flushed anyway. The percentage thresholds used to adjust the STLB size could be potentially dynamically adjusted as well, but this work chooses two reasonable values for simplicity.

This optimization enables the emulated STLB to be sized dynamically according to the working set of the running process and has the potential to increase the STLB hit rate and to reduce the STLB flush time—that is, taking the best of both worlds. However, it changes the STLB lookup, as the indexing hash value is not constant. The current hash value is placed in the structure where emulated processor states are kept (i.e., in the *CPUState* structure) at every context switch and is loaded from the *CPUState* structure at the time a memory operation needs to be translated, as shown in Figure 11. As shown in Figure 7, the dynamically sized STLB (STLB^{ds}) improves performance by 16.9% on average. Moreover, STLB^{ds} outperforms the fixed sized STLB^{4k} and STLB^{64k}. Additionally, with the STLB^{ds}, flushing on context switches is not as expensive even with frequent context switches, such as in the kernel boot and the multiprogrammed workloads.

3.4. Victim STLB

Implemented as a directly mapped hash table, the STLB suffers from conflict misses. Even when its size is dynamically adjusted, the STLB can suffer from conflict misses when the measured load is not high enough to trigger a doubling in size. Therefore, this work finds that introducing a victim STLB [Garibay Jr. et al. 1998] for fixed and dynamically sized STLBs provides significant benefits. A victim STLB holds translations evicted from the primary STLB on replacement. The victim STLB lies between the primary STLB and its refill path, and it is probed only on primary STLB misses. The victim STLB is generally of greater associativity, and for this reason it takes longer to look up the victim STLB than the primary STLB. However, probing the victim STLB is considerably faster than a full-page walk. The performance trade-off with a victim STLB is as follows. The victim STLB, with its higher associativity, can improve the

Fig. 12. Performance improvement with a victim STL B.

overall STL B hit rate by reducing conflict misses, thus resulting in fewer page table walks. Moreover, given its relatively small size, it does not adversely increase the STL B flush overhead. However, the victim STL B increases refill latency.

As Figure 12 shows, adding an eight-entry victim STL B to the baseline STL B²⁵⁶, denoted by STL B^{256+v}, improves performance by 11.1% on average, and by as much as 21.22% for 429.mcf. Adding an eight-entry victim TL B to STL B^{ds}, denoted by STL B^{ds+v}, proves more beneficial, improving performance by 22.9% on average, and by as much as 47.4% for 429.mcf. This is because the victim STL B can eliminate some of the page table walks resulting from STL B conflict misses when the STL B load is not high enough to double its size.

3.5. Set-Associative STL B

Set associativity has long been used to reduce conflict misses in hardware caches and TLBs [Chen et al. 1992]. Considering that, as Figure 4 shows, most misses in a directly mapped STL B are conflict misses, this section investigates set-associative STL B (saSTL B) implementations. Modern hardware TLBs usually are highly or fully associative [Chen et al. 1992], as hardware can simultaneously search through all ways in parallel. However, software has to search through the ways serially. Therefore, it is not clear whether increasing the STL B's associativity would be beneficial. On one side, associativity will reduce conflict misses, but on the other, it will increase lookup latency for *all* accesses.

This section considers four implementations of saSTL B s that differ in the way in which they search through a set's entries for a hit. First, search through the entries in the order in which they are stored in memory. Then, as in (1) but on a miss, reorder entries according to their hit count. Next, scan the entries in the order in which they were each most recently used. Finally, use vector extension instructions to search through multiple entries in parallel. Results are shown only for two-way and four-way set-associative STL Bs, as using higher associativity likely leads to worse performance—in other words, four-way saSTL B s underperform two-way saSTL B s, which underperform those that are directly mapped.

3.5.1. In-Order Lookups. As Figure 13 shows, the machine code to walk a four-way saSTL B builds on the code used for the directly mapped STL B. The lookup first hashes into a set and then checks the ways for a match one by one. Since all of the ways in a set are stored contiguously in memory, only one hash is needed and a constant can be added to find each of the remaining ways.

Fig. 13. Emulated four-way set-associative STLB walk.

Fig. 14. Set-associative STLB translation lookup time.

Although set associativity reduces STLB misses, significantly more time ends up being spent in STLB lookups, as shown in Figure 14. The figure includes measurements for STLBs with 256 entries that are organized as direct mapped (STLB²⁵⁶), two way (STLB^{128x2}), or four way (STLB^{64x4}). The figure includes measurements for other

Fig. 15. Set-associative STLb performance improvement.

configurations that subsequent sections explain. On average, STLB^{128x2} and STLB^{64x4} spend 15.7% and 63.9% more time in STLb translation than the direct-mapped STLb baseline, resulting in a 1.89% and 7.41% average performance degradation, respectively. Associativity increases STLb lookup time, as frequently accessed STLb translation entries are being installed into higher ways. Overall, as shown in Figure 15, performance suffers with the sASTLb s improving only slightly for the multiprogrammed workloads. In the interest of space, the remainder of this section considers two-way and four-way sASTLb enhancements.

3.5.2. Access Frequency-Based Reordering. To increase the chances that a matching entry is found early on STLb lookups, the entries can be reordered according to their hit frequency. A lightweight profiling technique is implemented to track the hit count per STLb entry and to reorder them accordingly. The implementation uses a counter per STLb entry, which it increments on a hit. This process increases the STLb lookup path by one instruction. On an STLb miss, an emulation code cache exit is triggered and the set's entries are reordered according to their hit counts (reordering on every STLb access would increase latency for all accesses considerably). As Figure 15 shows, this reordering ($\text{STLB}^{128x2+fo}$ and $\text{STLB}^{64x4+fo}$) improves performance, sometimes albeit only slightly. On average, $\text{STLB}^{128x2+fo}$ and $\text{STLB}^{64x4+fo}$ spend 13.3% and 37.7% more time in STLb translation than the direct-mapped STLb baseline, resulting in a 1.45% and 5.51% average performance degradation, respectively.

3.5.3. Relative Access Order Lookups. Checking the translations in the order in which they were most recently used may shorten the search path. Doing so requires keeping track of the relative access order of the entries within each set. A natural software implementation uses a linked list per set. The evaluated implementation keeps a table *hit-1* of per-set head pointers and augments every STLb entry with a *next-hit* pointer to the next in access order entry in the same set. Updating and following these pointers requires a considerable number of instructions. Figure 16 shows that on an STLb hit in a different way than the one pointed by the head pointer, two pointers need to be updated: the *hit-1* and the *next-hit* of the entry that hit.

Figure 15 also shows that reordering based on hit recency ($\text{STLB}^{128x2+ro}$ and $\text{STLB}^{64x4+ro}$) results, sometimes, in modest performance improvements, but overall it still hurts performance. On average, $\text{STLB}^{128x2+ro}$ and $\text{STLB}^{64x4+ro}$ spend 14.5% and 44.8% more time in STLb translation than the direct-mapped STLb baseline, resulting in a 1.65% and 6.54% average performance degradation, respectively. Although the presented technique reduces the number of executed instructions needed to find a

Fig. 16. Emulated four-way set-associative STLB walk with LRU.

matching entry, updating the recency information requires up to two instructions per hit. Additionally, the implementation uses two additional registers (*%rax* and *%rcx*) for the STLB lookup and thus increases register pressure around that frequently executed code, which in turn may increase instruction count when the guest instructions are translated into host-emulated ones.

3.5.4. SIMD Set-Associative Lookups. Using AVX 2.0 instructions [Firasta et al. 2008] on the x86 Haswell architecture [Jain and Agrawal 2013], the different ways of the TLB can be searched in parallel. Figure 17 shows the instruction sequence for a four-way sASTLB lookup that is generated using GCC intrinsics and then copied to the code cache. The four-way sASTLB is reorganized so that the translated addresses and offsets of the same set are laid out contiguously in memory, in separate groups. This way, all translation offsets in the same set can be cached in a single 64-byte cache block, and a single-load instruction can be used to fetch the translated addresses of the four entries of a set. As shown in Figure 14, this implementation reduces the STLB lookup time compared to the other four-way sASTLB implementations. However, compared to the direct-mapped STLB, lookup time is worse, and as a result, using SIMD instructions hurts performance by 4.43% on average (Figure 15). The SIMD two-way sASTLB is

Fig. 17. Emulated STLB SIMD walk.

expected to perform worse than SIMD four-way sASTLB, as it requires the same STLB lookup sequence while incurring more conflict misses due to lower associativity.

This section has shown that none of the four sASTLB implementations improves performance consistently. Moreover, on average, all hurt performance. Additional optimizations or a different approach is needed for a set-associative STLB to be beneficial. The rest of this work does not consider set-associative STLBs any further.

3.6. Translation Coalescing

Translation coalescing (XC) exploits a common scenario where two accesses happen to fall into the same page. Specifically, given a pair of effective addresses p and q off the same base register (i.e., $p = \text{Mem}(\text{OffsetA}, \text{BaseRegX})$ and $q = \text{Mem}(\text{OffsetB}, \text{BaseRegX})$), there is a good chance that both references will be on the same page if the difference $\text{abs}(\text{OffsetA} - \text{OffsetB})$ is small. In such a scenario, XC does only one translation with a modified operand length representing the region that encompasses both accesses, and computes the second address by adding an appropriate offset to the first translated address. For example, given two 64-bit accesses, one at $0x20(\%rsp)$ followed by another at $0x30(\%rsp)$, the first translation is done for $0x20(\%rsp)$ and a length of 24 bytes. The translated address for the $0x30(\%rsp)$ access is calculated by adding an offset of 16. The technique can be applied to longer access sequences.

To ensure that XC maintains correctness, a check must be made that the base register is not modified in the section of code between the coalesced accesses. To implement XC, the emulator tracks all memory accesses as well as their base registers and offsets in a translation block. When multiple instructions using the same base register are identified, XC generates appropriate code when the first memory access is emulated. If the translation happens to fall on a single page, the guest-to-host address translation offset is stored into the CPU structure. Subsequent instructions can perform address

Fig. 18. Translation coalescing performance improvement.

translation using a single add instruction as long as a valid offset is found in the CPU structure.

XC is particularly effective for stack accesses, and for function prologues and epilogues where sequences of pushes and pops are used to save and restore registers. However, the effectiveness of XC is constrained by translation unit length, as coalescing is not possible across different translation units. Moreover, since QEMU translates guest instructions a basic block at a time, coalescing is restricted to within a basic block. As shown in Figure 18, XC eliminates 7.9% of the STLB lookups, which translates to a 1.2% performance improvement on average.

3.7. Superpage TLB

Superpages, usually one to several orders of magnitude larger than regular pages, are used to extend the reach of TLB entries. In the x86 system used in this study, a regular page is 4KB and a superpage (hugepage on x86 Linux) is 2MB. Currently, QEMU's STLB has no notion of superpages: it records translations for 4KB pages, and the generated STLB lookup sequence always assumes 4KB pages. When a miss happens to a superpage, QEMU walks the guest page table and installs in the STLB a translation for the 4KB page chunk on which the miss happened. This way, the lookup sequence remains the same for all page sizes. However, this approach has two shortcomings. First, what would be a single page table walk for a superpage on the guest machine during emulation may require up to 512 page table walks, one per 4KB chunk. Second, a superpage that would require a single STLB entry in the guest machine could now require up to 512 entries under emulation and thus could hurt the STLB's hit rate.

This section considers three superpage STLB enhancements. The first reduces page table walks for superpages while still maintaining the same 4KB page lookup sequence for all accesses. The second introduces a separate superpage STLB that handles all superpage translations but is accessed in series with the base 4KB STLB. The third improves on the second by using profiling to tailor the lookup code so that the most likely page size per access is looked up first. In this section, 4K-entry 4KB STLB and 512-entry 2MB superpage STLB are used. Furthermore, attention is limited to the following workloads that do or can be made to use superpages: kernel boot that does use superpages and the Java workloads appropriately configured to use superpages. All other sections use the widely employed default Java configuration that does not explicitly use superpages.

Fig. 19. Superpage STLB performance improvement.

3.7.1. Avoiding Page Walks with a Secondary Superpage STLB. The first implementation introduces a superpage STLB, which it consults after a miss in the base 4KB page STLB and before the page table walk. After a page walk for a superpage, this implementation installs the translation for the superpage in the secondary STLB. Other than that, the default QEMU treatment of superpages persists: all lookups go through the 4KB STLB, and misses on superpages install a translation for the corresponding 4KB chunk. Page walks, however, are avoided on a hit to the secondary STLB. At the same time, the lookup sequence remains streamlined.

As shown in Figure 19 (the next section describes the additional techniques shown), a 512-entry superpage STLB alongside a 4K-entry 4KB STLB improves performance by 5.1% on average. The h2 entry, which uses a large Java heap backed by superpages, sees the largest performance benefit of 12.0%.

3.7.2. Using a Primary Superpage STLB. The secondary superpage STLB reduces page table walks but does nothing to relieve pressure from the 4KB STLB or to avoid incurring a primary STLB miss per 4KB chunk for superpages. This section promotes the secondary superpage STLB into a distinct superpage STLB. Superpage translations are installed only on this superpage STLB, relieving pressure from the 4KB STLB. The key challenge with this organization is that given a guest VA, it is not known in advance whether it belongs to a 4KB page or a superpage. Accordingly, a straightforward lookup strategy would probe the two STLBs in series. However, placing the 4KB STLB before the superpage STLB hurts every superpage hit, whereas the reverse lookup sequence hurts every 4KB page hit.

As shown in Figure 19, looking up the first 4K STLB and the 2M STLB second, denoted by 4K-2M, provides an average performance improvement of 6.5% over the baseline, which has no superpage STLB. Always looking up the 2M STLB and 4K STLB in series, denoted by 2M-4K, provides an average performance improvement of 4.8% over the baseline. As expected, probing the 4KB STLB is better on average, as 4KB pages are more common. However, in certain workloads, such as the Java ones, accesses to the much larger superpages are more common. It is for this reason that probing the superpage STLB first proves better sometimes.

3.7.3. Adaptive Lookup Sequence. An adaptive lookup sequence is presented that dynamically tailors the lookup code to probe the most likely to hit STLB first. Emulated memory instructions are profiled for the page size that they access, and their STLB lookup sequence is tailored accordingly. Algorithm 1 shows the page size profiling method used. A *struct pagesizerec* is allocated, on demand, for every emulated instruction when it misses in the STLB. The structure maintains a set of counters that record how often the instruction misses on a 4KB or a 2MB page. A hash table is used to allow quick lookups and updates of these structures. Every 255 STLB misses, the counter

ALGORITHM 1: Page Size Prediction Algorithm

```

1: ▷ update miss and profile miss counters
2: if is_4k_page(missaddr) then total_miss_4K++;
3: end if
4: if is_2m_page(missaddr) then total_miss_2M++;
5: end if
6: ▷ check profiling results periodically
7: if (total_miss_4K + total_miss_2M) % 255 == 0 then
8:   ▷ this instruction accesses only 4K or 2M pages
9:   if total_miss_4K == 0 then
10:    patch call to 2M – only STLB lookup;
11:    goto reset;
12:   end if
13:   if total_miss_2M == 0 then
14:    patch call to 4K – only STLB lookup;
15:    goto reset;
16:   end if
17:   ▷ this instruction accesses 4K and 2M pages.
18:   if total_profiled_miss_2M ≥ total_profiled_miss_4K then
19:    patch call to 2M – 4K STLB lookup;
20:    goto reset;
21:   end if
22:   if total_profiled_miss_4K > total_profiled_miss_2M then
23:    patch call to 4K – 2M STLB lookup;
24:    goto reset;
25:   end if
26:   reset :
27:   total_miss_2M = total_miss_4K = 0;
28: end if

```

values are compared, and the emulation code is patched accordingly. Four STLB lookup sequence variants are used, 4KB-only, 2M-only, 4K-2M, and 2M-4K, respectively, for memory instructions that access only 4K, only 2M, mostly 4K, and mostly 2M pages. The *struct pagesizerec* occupies 16 bytes in the tested host system, and thus host memory usage remains limited. The profiling mechanism initially assumes that a memory instruction will be 4KB-only, and only if it accesses 2MB pages will it inevitably trigger STLB misses and eventually be patched to an appropriate STLB lookup sequence. Algorithm 1 shows the profiling policy used in detail.

As shown in Figure 19, the profiling-based adaptive lookup offers an average performance improvement of 11.2% over the baseline in which no superpage STLB is used. Profiling works well because instructions do tend to access only one page size. Figure 20 shows a breakdown of all memory accessing instructions according to the page sizes that they access at runtime. For all studied workloads, most instructions access either 4KB or 2M pages, with only a small percentage of instructions accessing both 4KB and 2MB pages. Probing further, it is found that many of these 4K and 2M instructions are in the Linux kernel, and some of the instructions come from the operating system kernel code, such as those used to clear pages. Similar behavior has been observed in *_copy_from_user*, *_copy_to_user*, and *__memcpy* in the kernel. Overall, the translation time for these instructions represents a small fraction of the overall execution time.

Fig. 20. Page size distribution based on PC.

Fig. 21. Overall optimizations performance improvement.

4. PUTTING EVERYTHING TOGETHER

Thus far, the various optimizations were studied mostly in isolation. This section considers combining the various techniques and reports the resulting performance improvement. Although some of the optimizations are exclusive to others, many of the optimizations are complementary. This section presents performance results by combining some complementary techniques that were studied. Results are shown for some combinations that proved beneficial.

As Figure 21 shows, combining dynamic resizing, bump-the-STLB, victim, and translation coalescing (STLB^{ds+v+xc+bts}) performs the best, achieving an average performance improvement of 24.4% over the baseline 256-entry STLB. Furthermore, STLB^{ds+v+xc+bts} has very minimal requirements on the STLB flushing thread (i.e., an average of 0.42% of the time of emulation thread as shown in Figure 22). The performance improvements shown are over a modified version of QEMU that does not use inlined STLB lookup stubs. As Section 2.2 has shown, outlining these STLB lookup stubs and generating calls to a common instruction sequence for all emulated memory instructions provides an average 4.2% performance improvement. Therefore, the average performance improvement over the current release version of QEMU is even higher.

5. CONCLUSIONS

This work has shown that a significant amount of emulator time is spent in memory emulation. To the best of our knowledge, this is the first work studying memory translation emulation acceleration. This work quantitatively measured the time spent in

Fig. 22. Emulation thread versus flush thread time in dynamically sized STLB.

memory emulation and demonstrated that a DAT accounts for a significant portion of overall emulation time. Motivated by this observation, a series of techniques for improving STLB performance were investigated. Some of the techniques were inspired by optimizations applied to existing hardware TLBs. However, this work also observed that the design space of software-emulated TLBs is different from the design space of hardware TLBs, thus enabling optimizations that are different. More specifically, experiments found that not all hardware-inspired optimizations are suitable for software TLBs. For example, set associativity does not work well in an STLB, mainly because an STLB is searched serially and the benefits from reduced conflict misses are not enough to recover the additional time spent in the serial lookups. Others, such as a victim STLB, work very well. This work also found that software-specific techniques such as dynamically resizing the STLB and translation coalescing additionally worked well.

Overall, combining complementary hardware-inspired and software-enabled techniques, including dynamic resizing, bump-the-STLB, victim, and translation coalescing, achieved an average emulation performance improvement of 24.4% over a wide range of benchmarks and as much as 48.0% for a memory-intensive workload.

REFERENCES

- Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*. USENIX Association, Berkeley, CA, 41. <http://dl.acm.org/citation.cfm?id=1247360.1247401>.
- Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGPLAN Notices* 43, 3, 26–35. DOI: <http://dx.doi.org/10.1145/1353536.1346286>
- Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, 169–190. DOI: <http://dx.doi.org/10.1145/1167473.1167488>
- Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient memory virtualization for cross-ISA system mode emulation. In *Proceedings of the 10th ACM SIGPLAN / SIGOPS International Conference on Virtual Execution Environments (VEE'14)*. ACM, New York, NY, 117–128. DOI: <http://dx.doi.org/10.1145/2576195.2576201>
- J. Bradley Chen, Anita Borg, and Norman P. Jouppi. 1992. A simulation based study of TLB performance. *ACM SIGARCH Computer Architecture News* 20, 2, 114–123. DOI: <http://dx.doi.org/10.1145/146628.139708>

- Yufei Chen and Haibo Chen. 2013. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 207–218. DOI: <http://dx.doi.org/10.1145/2442516.2442537>
- Android Developers. 2011. What Is Android.
- Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. 2008. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*. Intel White Paper.
- Raul A. Garibay Jr., Marc A. Quattromani, and Douglas Beard. 1998. Address translation unit employing a victim TLB. U.S. Patent 5,752,274.
- John L. Hennessy and David A. Patterson. 2003. *Computer Architecture: A Quantitative Approach* (3rd ed.). Morgan Kaufmann, San Francisco, CA.
- John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4, 1–17.
- Bruce Jacob and Trevor Mudge. 1998. Virtual memory in contemporary microprocessors. *IEEE Micro* 18, 4, 60–75.
- Tarush Jain and Tanmay Agrawal. 2013. The Haswell microarchitecture—4th generation processor. *International Journal of Computer Science and Information Technologies* 4, 3, 477–480.
- JVM JVMTI. 2005. Tool Interface, v1. 0.
- Ehsan K. Ardestani and Jose Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'19)*. 448–459.
- Kevin P. Lawton. 1996. Bochs: A portable PC emulator for Unix/X. *Linux Journal* 1996, 29es, Article No. 7. <http://dl.acm.org/citation.cfm?id=326350.326357>.
- John Levon. 2004. *OProfile Manual*. Victoria University of Manchester.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 190–200. DOI: <http://dx.doi.org/10.1145/1065010.1065034>
- Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2, 50–58. DOI: <http://dx.doi.org/10.1109/2.982916>
- Bill Ogden. 2013. *IBM System z Personal Development Tool: Volume 2 Installation and Basic Use*. IBM Redbooks.
- Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, New York, NY, 1050–1055. DOI: <http://dx.doi.org/10.1145/2024724.2024954>
- Georgios Portokalidis, Asia Slowinska, and Herbert Bos. 2006. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys'06)*. ACM, New York, NY, 15–27. DOI: <http://dx.doi.org/10.1145/1217935.1217938>
- Xin Tong, Jack Luo, and Andreas Moshovos. 2013. QTrace: An interface for customizable full system instrumentation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, Los Alamitos, CA, 132–133.

Received April 2014; revised September 2014; accepted November 2014